From Free Categories to Flip Fields: The Spacetime Semiotics of Category Theory and HoTT in a Running Program

Douglas C. Youvan

doug@youvan.com

August 11, 2025

Category Theory (CT) and Homotopy Type Theory (HoTT) are typically introduced in purely symbolic terms, divorced from the physical realities of computation. Yet, when implemented in a program like core7.py, these structures acquire a concrete embodiment in memory and processor state. This paper follows the emergence of CT/HoTT constructs inside core7.py from their algebraic form to their physical realization as fields of bit flips in spacetime. Objects persist as stable clusters of bits in RAM, morphisms appear as pointer-encoded adjacency relations, and lemmas become idempotent memory regions that fill cycles in the computational graph. Normalization of paths — replacing long pointer chains with shorter ones — reduces toggle activity, narrows causal cones, and contracts the hardware footprint of proofs. We introduce measurable quantities such as toggle density, semantic compression ratio, and causal cone contraction to link mathematical simplification with physical economy. Using a topos-theoretic lens, we interpret the run as a presheaf over time, with lemma discovery as a natural transformation. The complete core7.py source code is provided in an appendix for transparency and replication.

Keywords: Category Theory, Homotopy Type Theory, computational topology, presheaf, effective topos, lemma rewriting, minimal description length, semiotics, bit flips, causal cone, toggle density, semantic compression, pointer chasing, hardware embodiment, emergent law. 38 pages. A collaboration with GPT-5.
CC4.0.

Note: The core7.py source code is in the final appendix.

I. Introduction

A. Motivation: moving from textbook CT/HoTT to its digital embodiment in hardware

Category Theory (CT) and Homotopy Type Theory (HoTT) are traditionally taught and studied as symbolic disciplines: diagrams on paper, algebraic definitions, and proofs existing in a purely abstract space. While this abstraction is essential for their generality, it can obscure the fact that when these ideas are implemented in software, they acquire a tangible existence in physical hardware. Inside a running program, objects, morphisms, and higher cells are not just symbols — they are bit patterns stored in memory, linked by pointers, and manipulated by the processor through sequences of bit flips over time. This paper addresses the question: what does CT/HoTT *look like* when it lives as data and control flow in a real machine?

B. The case study: core7.py as an emergent category engine

core7.py is a stochastic graph-growth and rewriting system that instantiates a free category on a random generator graph and gradually quotients it by discovered 2-cells. Its rules are minimal: compose paths, compare endpoints, record equalities, and bias toward shorter representatives using a minimal description length (MDL) heuristic. Over time, the system develops stable categorical laws — such as commuting diagrams — without these laws being explicitly programmed. This makes it an ideal platform to study how categorical structure emerges from computation.

C. Why “discovered, not invented” applies both to mathematics and machine-realized structure

A longstanding philosophical position holds that mathematics exists independently of human minds, waiting to be discovered rather than invented. core7.py operationalizes this viewpoint: the program does not predefine the specific commuting diagrams it will find, yet they emerge reliably across different runs. The underlying structures — path composition, identity of endpoints, preference for shorter forms — are inherent in the system’s rules, not authored ad hoc. Thus, just as humans “discover” categorical laws in the abstract, the machine “discovers” them in its own evolving data universe.

D. Overview of contributions: linking categorical constructs to physical toggle dynamics

This paper makes three linked contributions:

1. **Algebra-to-hardware mapping** — We map standard CT/HoTT constructs (objects, morphisms, 2-cells, normalization) to their in-memory representations as persistent bit-clusters, pointer-encoded adjacency relations, and idempotent update regions.
2. **Quantitative metrics** — We introduce measurable quantities, including toggle density ρ(addr, t), semantic compression ratio (k_L / k_R), and causal cone contraction ΔC, to link algebraic simplification with reduced hardware activity.
3. **Topos-theoretic framing** — We interpret the evolving state as a presheaf over time, with lemma application as a natural transformation, bridging physical computation and categorical semantics.

By following core7.py from abstract rules to physical execution, we demonstrate that CT/HoTT is not only a mathematical theory but also a physically instantiated process in hardware.

II. From Source Code to Mathematical Presentation

A. The free category on a directed multigraph in memory

At its core, core7.py builds and maintains a directed multigraph stored in Python dictionaries and lists. Each node in the graph represents an *object* in the category; each directed edge represents a *generating morphism*. The program starts with a small number of objects and morphisms, then stochastically adds new objects and edges over time. This structure is the *free category* on the current generator graph: all finite sequences of generating morphisms are valid composable paths, and no identifications are made except those implied by the program's discovery of equalities (2-cells). In memory, this category is not stored as an explicit set of all paths; instead, the paths are constructed on demand by traversing the adjacency lists of the multigraph.

B. Morphism composition as path concatenation in lists

In the free category, composition of morphisms is simply concatenation of arrows. In core7.py, a morphism path is represented as a Python list of morphism IDs. Composing f: A → B and g: B → C corresponds to creating a new list by appending the ID of g to the end of the list representing f. This means that composition is an O(1) operation for appending a single morphism, but an O(k) operation when creating longer concatenations, where k is the length of the resulting path. These lists serve as the program's operational equivalent of formal words in the free category.

C. 2-cells as commuting diagram closures (triangles, squares, polygons, path-vs-path)

The engine includes routines to detect when two distinct paths between the same objects should be considered equal. Four closure types are tested:

1. **Triangle closure** — Detects that a two-step path A → B → C is equivalent to an existing one-step arrow A → C, or mints the direct arrow if missing, then records a 2-cell equating them.
2. **Square closure** — Similar, but compares a three-step path A → B → C → D with a one-step arrow A → D.
3. **Polygon closure** — Compares a longer random walk from A to Z against an existing direct arrow A → Z.
4. **Path-vs-path closure** — Finds two different random paths between the same endpoints and equates them directly.
   When an equality is detected, a 2-cell is recorded as evidence, just as in higher category theory a 2-cell witnesses the commutativity of a diagram.

D. Rewriting lemmas and MDL bias toward canonical forms

Each 2-cell leads to a rewrite rule (lemma) that replaces the longer path with the shorter one. These lemmas are stored in a dictionary keyed by the longer path, with the value being the canonical shorter path. The preference for shorter forms is governed by a minimal description length (MDL) bias: the program chooses equalities that reduce overall path length and redundancy in the morphism

representation. This bias gradually pushes the category toward a more compact presentation, in which many distinct syntactic paths collapse to the same semantic morphism.

E. Categorical summary: quotienting the free category by an emergent congruence

In standard category theory, a presentation of a category consists of generators (objects and morphisms) and relations (equalities between morphisms). core7.py begins with only the generators and no relations, thereby constructing the free category on its graph. As the run proceeds, each discovered lemma corresponds to a relation that identifies two morphisms. The set of all such relations generates a congruence on the free category, and the program's current universe at any epoch is effectively the quotient of the free category by this emergent congruence. This quotient is not predetermined; it evolves as new equalities are discovered, making the program a dynamic *presentation finder* for small categories.

III. HoTT Interpretation in Runtime Terms

A. Paths as identity type inhabitants

In Homotopy Type Theory (HoTT), given two points x and y in a type A, an *identity type* Id_A(x, y) represents the type of proofs (or paths) that x and y are equal. In core7.py, a *path* — represented as a list of morphism IDs — is the computational analogue of such an inhabitant. If a path starts at object A and ends at object B, then the program treats it as an explicit witness of a morphism from A to B. The set of all such paths from A to B is the operational equivalent of Id_A(A, B). Unlike in pure HoTT, these paths are finite sequences generated by traversal, not infinitely large proofs, but they retain the role of concrete evidence for a relationship between two points.

B. 2-cells as operational homotopies

In HoTT, a homotopy between two paths p and q from A to B is itself a higher-dimensional proof: evidence that the proofs p and q are equal. In core7.py, when

the program records a 2-cell, it is storing exactly this kind of evidence. The 2-cell says: "These two distinct computational paths produce the same morphism from A to B." In the data structure, the 2-cell becomes a rewrite lemma mapping the longer path to the shorter one, but semantically it is the program's record of a homotopy between two inhabitants of the same identity type.

C. Partial infinity-groupoid fragment realized as a rewriting system

HoTT interprets types as ∞-groupoids, where points are objects, 1-paths are morphisms, 2-paths are homotopies between morphisms, and so on. core7.py implements a 2-truncated fragment of such a structure:

- **Level 0**: Objects in the category.
- **Level 1**: Morphisms (paths) between objects.
- **Level 2**: 2-cells (lemmas) as homotopies between morphisms.
  Higher cells are not explicitly represented. The rewriting system serves as the computational mechanism for applying these 2-cells, normalizing longer proofs into shorter canonical forms. The resulting effect is similar to *normalization by evaluation* in type theory, where terms are reduced to canonical representatives before comparison.

D. Limitations: absence of enforced univalence or higher coherence

The program does not attempt to enforce the *univalence axiom*, which in HoTT identifies equivalences between types with equalities of types. Nor does it build or check higher coherence conditions (e.g., pentagon or triangle identities for associators in higher categories). Instead, core7.py discovers only those equalities that arise stochastically during exploration. If univalence-like or coherence-like properties emerge, they do so empirically, not by construction. This means that the program's categorical universe may lack some structural guarantees that a fully axiomatized HoTT setting would require, but it also allows for the study of how such structures might arise naturally from minimal computational rules.

IV. The Digital Topology of Program State

A. Objects as persistent bit-clusters in RAM

Each categorical object in core7.py has a direct physical representation in the heap as a Python integer, string, or other immutable key stored in a dictionary. These structures occupy specific ranges of memory addresses that change infrequently, making them *persistent bit-clusters*. Their internal bit patterns toggle rarely once allocated — primarily when referenced or copied — giving them a low local toggle density over long time intervals. In the digital topology, these clusters function as *0-dimensional anchor points* in the computational graph, providing stable endpoints for morphisms and paths.

B. Morphisms as pointer-encoded adjacency relations

A morphism record links a source object to a target object, typically by storing two references (pointers) to the objects' locations in memory along with a label or ID. This creates a *pointer-encoded adjacency* between two persistent bit-clusters. In physical terms, composition of morphisms involves chasing these pointers through memory: source → morphism record → target. This pointer structure defines the program's *1-skeleton*, the graph of edges that connects the object-vertices in the digital topology.

C. Lemma store as an idempotent update region

When a new 2-cell is discovered, core7.py stores it as a rewrite lemma mapping a longer path to a shorter one. This lemma is placed in a dictionary keyed by the long path, with the value being the canonical short path. Once written, this memory region becomes *idempotent*: subsequent applications of the lemma do not change its content, only trigger reads. In the digital topology, this is equivalent to filling a 2-cell into the 1-skeleton, making a previously nontrivial loop contractible. Physically, the lemma store's toggle density drops sharply after insertion, reflecting a shift from write-heavy to read-heavy access.

D. Normalization as shrinkage of pointer-chasing trails

Before a lemma is applied, traversing a path of length k requires k pointer dereferences, each with an associated toggle cost from CPU registers, caches, and possibly DRAM. After normalization, the same morphism is represented by a path

of length k′ where k′ < k. The result is a measurable reduction in both pointer dereferences and total bit flips for equivalent computations:

$\Delta\text{flips} \approx E(\text{flips} \mid k) - E(\text{flips} \mid k')$

Here, $E(\text{flips} \mid k)$ represents the expected number of bit flips needed to traverse a path of length k. This shrinkage is both a computational optimization — fewer steps to reach the same result — and a topological contraction, shortening the "distance" between two objects in the program's 1-skeleton without altering its categorical semantics.

V. Spacetime Dynamics of 1's and 0's

A. Layers of propagation: core, cache hierarchy, DRAM, I/O

When core7.py runs, the categorical structures described in earlier sections are embodied in physical events: 1's and 0's flipping in the processor and memory subsystems. These events occur in distinct layers, each with its own propagation speed, latency, and locality:

1. **Core logic layer** — Billions of bit flips per second occur in CPU registers, arithmetic-logic units, and control logic. These are the fastest transitions, propagating at a significant fraction of the speed of light within the silicon.
2. **Cache hierarchy** — L1, L2, and L3 caches store recently accessed data. Bit flips here occur in bursts when cache lines (typically 64 bytes) are read or written.
3. **DRAM layer** — Main memory operations are slower and more spatially extended: an entire DRAM row activates, sense amplifiers fire, and data bursts across the memory bus.
4. **I/O layer** — Writes to disk or network, such as periodic report outputs, produce sporadic toggle bursts over I/O interconnects.

B. Toggle density as a spacetime field

We can define a toggle density function over address-time spacetime:

ρ(addr, t) = number of bit flips per second in a small address window at time t

Regions with high ρ are computational "hotspots." In core7.py, these typically occur in:

- Dictionaries storing objects and morphisms,
- The lemma store,
- The pseudorandom number generator state used for random walks and closure tests.

C. Causality cones from bit flips to dependent state changes

Each bit flip at ($addr_o$, t_o) can influence a set of later flips — its *causal future*. For example, a pointer dereference in a morphism record may lead to:

- Cache lookup in L1,
- Possible miss and fetch from L2 or L3,
- Possible DRAM access,
- Update to an internal register for subsequent computation.
  The set of flips reachable in this manner forms a causal cone whose temporal depth and spatial width depend on access patterns and cache performance.

D. Digital Alexandrov topology: partial order of events by influence

We can define a partial order on events: $e_1 \prec e_2$ if e_1 can causally influence e_2. The open sets in the Alexandrov topology are then causal futures of events. Lemma application contracts these futures: replacing a long path with a shorter one removes intermediate pointer dereferences and their downstream effects, shrinking the cone's volume in spacetime.

E. Toggle economics and lemma gains

Let k be the length of a morphism path. The expected toggle cost is:

$E(\text{flips} \mid k) \approx k \cdot c_{walk} + c_{hash} + c_{miss} \cdot p_{miss}(k)$

where:

- c_{walk} = flips per pointer dereference,
- c_{hash} = flips for hashing and key comparison,
- c_{miss} = flips on a cache miss,
- $p_{miss}(k)$ = probability of a cache miss for path length k.

When a lemma shortens a path from k to k′ (with $k' < k$), the toggle saving is:

$\Delta\text{flips} \approx E(\text{flips} \mid k) - E(\text{flips} \mid k')$

This represents a literal reduction in hardware work for the same categorical computation.

F. Interpretation

In spacetime terms, the evolution of core7.py is a reorganization of its flip field. Early in a run, the program explores longer paths, producing larger causal cones and higher toggle densities. As lemmas accumulate, paths shorten, toggle density decreases in key memory regions, and causal cones contract. This is not just a software-level optimization — it is a physical compression of computational activity in time and space, directly mirroring the algebraic compression in the category's presentation.

VI. Semiotics Inside the Machine

A. Peircean triad: icons, indices, and symbols in program state

Charles Sanders Peirce's triadic model of the sign — *icon*, *index*, *symbol* — provides a useful lens for interpreting the meaning of program state inside core7.py:

- **Icons**: Plots of object counts, morphism counts, and compression scores are icons in the strict Peircean sense — they *resemble* the growth and stabilization of categorical structure.
- **Indices**: Pointers in memory are indices, physically connected to the objects they reference. A lemma's presence in the store is an index of a past discovery event; its use in rewriting is causally linked to that stored bit pattern.
- **Symbols**: Object names, morphism IDs, and path tuples are symbols — arbitrary but conventional encodings whose interpretation depends on the system's internal rules.

B. Control flow river vs. heap mutation river

The machine's toggles organize into two dominant streams:

1. **Control flow river** — Instruction decoding, branching, and looping in the Python interpreter; here, 1's and 0's represent sequencing logic, not categorical data.
2. **Heap mutation river** — Writes to the dictionaries and lists that store the categorical universe; here, toggles directly correspond to adding objects, morphisms, and lemmas. The heap mutation river is where CT/HoTT semantics is physically realized in memory structures.

C. Stable attractors as laws

Once a lemma is inserted, it becomes a *stable attractor* in state space:

- It persists across epochs,
- It repeatedly causes the same rewriting when its left-hand side path is encountered,
- It reduces both toggle cost and path length in future computations.
  In semiotic terms, the lemma functions as a *law-symbol*, compressing many distinct syntactic forms into a single semantic identity.

D. Semiotic economy and compression

The minimal description length (MDL) heuristic in core7.py is a machine analog of semiotic economy — the tendency to prefer shorter, more efficient signs. If a lemma maps a left-hand side of length k_L to a right-hand side of length k_R, the *semantic compression ratio* is:

compression_ratio ≈ k_L / k_R

A higher ratio reflects both syntactic efficiency in the categorical presentation and physical efficiency in hardware activity.

E. The interpretant as process

In Peirce's model, the *interpretant* is the process that links signs to their meaning. In core7.py, the interpretant is the runtime rewrite engine coupled with the MDL bias. This interpretant:

1. Matches symbolic paths against stored lemmas (sign recognition),
2. Rewrites them to canonical shorter paths (meaning enactment),
3. Updates statistics such as compression score (meta-signs about meaning efficiency).

F. Bridging to Category Theory and HoTT

From a CT/HoTT perspective:

- **Sign** = a path (sequence of generating morphisms),
- **Object** = the morphism it denotes in the current quotient category,
- **Interpretant** = the rewrite procedure that applies discovered 2-cells to reduce the path to a canonical form.
  In this mapping, the semiotics of bit flips and pointer manipulations is *isomorphic* to the algebraic manipulation of morphisms: the physical hardware operations are not just implementing CT/HoTT, they *are* CT/HoTT instantiated in a digital medium.

VII. Mathematical Formulations for Toggle Economics

A. Expected toggle budget per composite

The execution cost of traversing or composing a morphism path can be modeled as the number of bit flips it causes in hardware. Let k be the length of the path (number of morphisms composed). The expected number of flips is:

$$E(\text{flips} \mid k) \approx k \cdot c_{\text{walk}} + c_{\text{hash}} + c_{\text{miss}} \cdot p_{\text{miss}}(k)$$

where:

- c_{walk} = average number of flips per pointer dereference in the path,
- c_{hash} = flips for hashing and key comparison when looking up lemmas,
- c_{miss} = flips caused by a cache miss,
- $p_{\text{miss}}(k)$ = probability of at least one cache miss during traversal of a path of length k.

This simple linear-plus-overhead model reflects both sequential pointer walking and random-access penalties.

B. Lemma-induced toggle savings

When a lemma rewrites a path from length k to a shorter length k′ (with $k' < k$), the expected toggle saving is:

$$\Delta\text{flips} \approx E(\text{flips} \mid k) - E(\text{flips} \mid k')$$

A positive Δflips means fewer total bit flips are required to compute the morphism, translating algebraic simplification into direct physical efficiency.

C. Semantic compression ratio

The *semantic compression ratio* of a lemma measures the degree to which it shortens its left-hand side (LHS) path:

$$\text{compression_ratio} \approx k_L / k_R$$

where k_L is the length of the LHS path and k_R is the length of the right-hand side (RHS) path. Ratios greater than 1 indicate a net shortening. This metric has

both syntactic significance (shorter expressions in the categorical presentation) and physical significance (less work for the hardware).

D. Toggle density reduction

Using the toggle density function defined in Section V:

ρ(addr, t) = flips per second in a small address window at time t

we can measure the effect of a lemma on the memory regions it affects. Let ρ_before be the average density in a region before lemma insertion and ρ_after be the average afterward:

Δρ = ρ_before − ρ_after

A positive Δρ indicates reduced local toggle activity due to the lemma's shortening effect.

E. Causal cone contraction

Each path traversal creates a causal cone of dependent flips in time and space. Let C(k) be the average number of causally dependent flips for a path of length k. When a lemma shortens a path from k to k', the reduction in cone size is:

ΔC ≈ C(k) − C(k')

Since k' < k, ΔC is expected to be positive, representing a narrowing of the computation's spacetime footprint.

F. Interpretation

These formulations bridge abstract category simplification and tangible machine behavior:

- Algebraically, lemma discovery adds new relations to the category's presentation, reducing morphism length in canonical form.
- Computationally, it reduces the number of pointer dereferences, lookups, and cache interactions.
- Physically, it lowers toggle density, shrinks causal cones, and conserves energy.

In short, every algebraic gain in presentation compactness has a measurable physical analog in the machine's operation.

VIII. A Topos-Theoretic View of Execution

A. Effective topos: program state as computable sets and maps

The *effective topos* is the categorical setting where objects are computable sets and morphisms are partial recursive functions between them. In this framework, the in-memory state of core7.py can be seen as an evolving object in the effective topos:

- **Objects in the category** (in the CT sense) are represented in hardware as finite bit strings, stored in Python's object memory.
- **Morphisms in the category** are implemented as computable maps between these finite encodings, realized in Python functions or C-level runtime routines.
- **2-cells** (lemmas) are additional computable relations, mapping one representation of a morphism to another, shorter one.

Thus, the entire runtime can be understood as a diagram in the effective topos, where computation corresponds to composing morphisms in this category of computable entities.

B. Presheaf over discrete time

Let T be a category whose objects are discrete time steps (epochs) t = 0, 1, 2, … and whose morphisms are successor maps t → t+1. The state of core7.py at time t — including its objects, morphisms, and lemmas — forms the set F(t). The execution is then modeled as a *presheaf*:

F : T^op → Set

where F(t) is the categorical presentation at epoch t, and F(t → t+1) is the restriction (or forgetting) map from the later state back to the earlier state. This

makes the run a time-indexed family of states connected by morphisms that preserve what was already known.

C. Causal structure and sheaf conditions

In practice, computation does not unfold strictly in global discrete steps; it has a partial order induced by causality. Let P be the poset of events in the run, ordered by "can influence." The Alexandrov topology on P defines open sets as causal futures of events. A *sheaf* over this topology assigns to each open set U a set of categorical data (objects, morphisms, lemmas) available in that region, with restriction maps to smaller causal regions. The sheaf condition — that local compatible data glue to form a global section — corresponds to the program's ability to merge partial knowledge from separate event regions into a coherent categorical state.

D. Colimits over seeds in batch runs

The batch runner in core7_batch.py executes multiple independent runs (different random seeds). Each run produces its own presheaf F_i : T^op → Set. After execution, we can combine these by taking the *colimit* over the index category of seeds:

colim F_i

This aggregates categorical structures from all runs into a single presentation. Persistent lemmas — those appearing in many runs — correspond to categorical laws that are stable across the ensemble. Colimits here act as an empirical analogue of intersection-of-invariants across multiple "possible worlds."

E. Interpretation

This topos-theoretic perspective unifies the abstract and the physical:

- The program's *hardware states* are elements of computable sets.
- The *execution trace* is a presheaf over a time or causality base.
- *Lemma discovery* is a natural transformation on this presheaf, simplifying its fibers (per-time states).

- Batch colimits merge separate universes into a single, more law-rich category.

In this view, core7.py is not merely executing instructions — it is evolving as a functor into the effective topos, with emergent structure driven by computational discovery.

IX. Discussion

A. Discovered, not invented

The central philosophical position underscored by this work is that the categorical and homotopical structures emerging in core7.py are *discovered*, not *invented*. The program's code does not dictate specific lemmas or commuting diagrams — it only establishes a minimal generative framework: compose paths, detect equal endpoints, and prefer shorter forms. The resulting structures arise naturally from these rules and are reproducible across runs. This parallels the human experience of mathematics, where new theorems are "found" in a landscape defined by axioms rather than arbitrarily authored. In this sense, core7.py is not merely a simulation of CT/HoTT but a computational experiment in uncovering preexisting mathematical forms.

B. Multi-level correspondence

The system exhibits a tight correspondence between multiple descriptive levels:

- **Algebraic**: The quotienting of the free category by an emergent congruence.
- **Computational**: Reduction of pointer dereferences, fewer lemma lookups, shorter path lists.
- **Physical**: Lower toggle density ρ(addr, t), reduced causal cone size ΔC, and smaller flip counts Δflips for equivalent computations.
- **Semiotic**: Lemmas as persistent law-symbols, compression as semiotic economy, rewriting as meaning enactment.
  These levels are not independent; an improvement in one (e.g., algebraic shortening) propagates to all others.

C. Measurement and verification

The physical correlates of emergent laws can be quantitatively verified. For a lemma rewriting $k_L \rightarrow k_R$, we can measure:

- **Semantic compression ratio**: $\text{compression_ratio} \approx k_L / k_R$
- **Toggle saving**: $\Delta\text{flips} \approx E(\text{flips} \mid k_L) - E(\text{flips} \mid k_R)$
- **Toggle density change**: $\Delta\rho = \rho_{\text{before}} - \rho_{\text{after}}$
- **Causal cone contraction**: $\Delta C \approx C(k_L) - C(k_R)$
  Consistency between algebraic gains and physical savings validates the claim that mathematical simplification has a direct material signature in hardware execution.

D. Implications for AI and formal systems

The core7.py case study suggests that minimal stochastic systems can uncover meaningful mathematical structure without explicit symbolic reasoning. This has implications for artificial intelligence:

- AI systems could be designed to *discover* higher-dimensional categorical or homotopical structures by embedding simple generative rules and bias toward compression.
- Such systems might yield formal laws whose physical execution footprint reflects their mathematical efficiency, creating a dual optimization loop — algebraic and physical.
- This approach could challenge traditional boundaries between “formal mathematics” and “machine execution,” pointing toward AI architectures that inherently embody the structures they manipulate.

X. Conclusion

A. Summary of CT/HoTT embodiment in hardware spacetime

This study has traced the life of Category Theory (CT) and Homotopy Type Theory (HoTT) concepts inside a running computational system, from abstract formalism

to physical instantiation. In core7.py, objects are persistent bit-clusters in RAM, morphisms are pointer-encoded adjacency relations, and lemmas are idempotent memory regions that fill loops in the category's 1-skeleton. Path normalization — algebraically a quotienting of the free category by new 2-cells — physically manifests as reduced pointer traversal, lower toggle density, contracted causal cones, and smaller total bit flip counts. In this sense, CT/HoTT is not only symbolically represented but materially embodied in the spacetime dynamics of the machine's execution.

B. Potential instrumentation for empirical measurement
Several measurable quantities make this embodiment testable:

- **Toggle density**: ρ(addr, t) = flips per second in a memory window, before and after lemma application.
- **Semantic compression ratio**: compression_ratio ≈ k_L / k_R for lemma rewrites.
- **Toggle saving**: Δflips ≈ E(flips | k_L) − E(flips | k_R).
- **Causal cone contraction**: ΔC ≈ C(k_L) − C(k_R).
  Instrumentation at the memory-access and cache-event levels could directly confirm the correlation between algebraic simplification and hardware efficiency.

C. Broader relevance to AI, formal systems, and philosophy of mathematics
By demonstrating that a minimal generative system can discover and embody mathematical structure without explicit symbolic guidance, core7.py points toward AI architectures that merge formal reasoning with physical optimization. Such systems would not just *model* mathematics but *live inside it*, with every discovered law reflected in a measurable reduction of physical computation cost. Philosophically, this blurs the line between mathematical discovery by humans and by machines, supporting the view that mathematics exists independently, awaiting discovery by any process — biological or digital — capable of navigating its landscape.

XI. References

1. Awodey, S. (2010). *Category Theory* (2nd ed.). Oxford University Press.
2. Leinster, T. (2014). *Basic Category Theory*. Cambridge University Press.
3. Mac Lane, S. (1998). *Categories for the Working Mathematician* (2nd ed.). Springer.
4. Riehl, E. (2017). *Category Theory in Context*. Dover Publications.
5. Univalent Foundations Program. (2013). *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study.
6. Lurie, J. (2009). *Higher Topos Theory*. Princeton University Press.
7. Lawvere, F. W., & Schanuel, S. H. (2009). *Conceptual Mathematics: A First Introduction to Categories* (2nd ed.). Cambridge University Press.
8. Peirce, C. S. (1931–1958). *Collected Papers of Charles Sanders Peirce* (Vols. 1–8). Harvard University Press.
9. Goguen, J. A. (1991). "Categories of Signification." In *Applications of Category Theory to Computer Science*, London Mathematical Society Lecture Note Series, 177, 1–52. Cambridge University Press.
10. Abramsky, S., & Tzevelekos, N. (2011). "Introduction to Categories and Categorical Logic." In *New Structures for Physics*, Lecture Notes in Physics, 813, 3–94. Springer.
11. Barwise, J., & Seligman, J. (1997). *Information Flow: The Logic of Distributed Systems*. Cambridge University Press.
12. Heller, A., & Streicher, T. (1992). "Semantics of Type Theory." In *Handbook of Logic in Computer Science* (Vol. 2). Oxford University Press.
13. Yau, D., & Johnson, M. W. (2022). *Homotopy Type Theory: The Logic of Space*. Cambridge University Press.
14. Ghrist, R. (2014). *Elementary Applied Topology*. Createspace Independent Publishing.

15. Fong, B., & Spivak, D. I. (2019). *Seven Sketches in Compositionality: An Invitation to Applied Category Theory*. Cambridge University Press.

16. Vicary, J. (2012). “Higher Categories for the Working Physicist.” *Lecture Notes*, University of Oxford.

17. Abramsky, S., & Coecke, B. (2008). “Categorical Quantum Mechanics.” In *Handbook of Quantum Logic and Quantum Structures*, 261–323. Elsevier.

18. Youvan, D. C. (2025). *Local Truth in Categorical Simulations: GenesisBoard as a Laboratory for Domain-Bounded Coherence*. August 2025. DOI: 10.13140/RG.2.2.11503.96164

19. Youvan, D. C. (2025). *Emergence of Higher-Order Structure in Category-Theoretic Graph Growth: Experimental Evidence from the GenesisBoard Framework*. August 2025. DOI: 10.13140/RG.2.2.21360.57606

20. Youvan, D. C. (2025). *GenesisBoard/core7.py: Experimental Documentation and Categorical Analysis*. ResearchGate Preprint, 2025.

Appendix: core7.py

```
#!/usr/bin/env python3
# core7.py — CT/HoTT aether engine with heartbeat + analysis (Windows-friendly)

import argparse, json, random, time, datetime, sys, math
from pathlib import Path
from typing import Dict, List, Tuple, Optional

VERSION = "core7 v2025-08-11 (clean)"

# ---------------------------- State types ----------------------------

State = Dict[str, object]
Morph = Tuple[str, str, str]   # (src, dst, name)

# ---------------------------- Helpers ------------------------------

def _new_run_dir() -> Path:
    now = datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
    root = Path.cwd() / "runs"
    root.mkdir(parents=True, exist_ok=True)
    # make collisions impossible if many children start at the same second
    base = root / f"run-{now}-{os_pid()}"
    n = 0
    while True:
        p = base if n == 0 else root / f"{base.name}-{n:03d}"
```

```
        try:
            p.mkdir(parents=True, exist_ok=False)
            return p
        except FileExistsError:
            n += 1

def _open_append(path: Path):
    path.parent.mkdir(parents=True, exist_ok=True)
    # newline="\n" prevents CRLF mishaps inside JSONL
    return open(path, "a", encoding="utf-8", newline="\n")

def os_pid() -> int:
    try:
        import os
        return os.getpid()
    except Exception:
        return int(time.time() % 100000)

# --------------------------- World ops ---------------------------

def _next_object_name(n: int) -> str:
    name = ""
    n += 1
    while n > 0:
        n, r = divmod(n - 1, 26)
        name = chr(65 + r) + name
```

```
    return name


def aether_bootstrap(state: State, add_object_p: float) -> None:
    objs: List[str] = state["objects"]  # type: ignore
    mor: List[Morph] = state["morphisms"]  # type: ignore
    if not objs:
        objs.append("A")
    if len(objs) == 1 and random.random() < add_object_p:
        objs.append("B")
    if len(mor) == 0 and len(objs) >= 2:
        a, b = objs[0], objs[1]
        mor.append((a, b, "f1"))

def mint_object(state: State) -> str:
    objs: List[str] = state["objects"]  # type: ignore
    name = _next_object_name(len(objs))
    objs.append(name)
    return name

def mint_morphism(state: State) -> Morph:
    objs: List[str] = state["objects"]  # type: ignore
    mor: List[Morph] = state["morphisms"]  # type: ignore
    if len(objs) < 2:
        return ("", "", "")
    import random as _r
    a, b = _r.sample(objs, 2)
```

```
    name = f"m{len(mor)+1}"
    mor.append((a, b, name))
    return (a, b, name)

def morphisms_from_to(state: State, src: str, dst: str) -> List[Morph]:
    mor: List[Morph] = state["morphisms"]  # type: ignore
    return [m for m in mor if m[0] == src and m[1] == dst]

def adjacency(state: State) -> Dict[str, List[Morph]]:
    adj: Dict[str, List[Morph]] = {}
    for (s, t, n) in state["morphisms"]:  # type: ignore
        adj.setdefault(s, []).append((s, t, n))
    return adj

def random_walk(state: State, start: str, max_len: int) -> Optional[Tuple[List[str], str]]:
    adj = adjacency(state)
    cur = start
    path: List[str] = []
    import random as _r
    steps = _r.randint(1, max_len)
    for _ in range(steps):
        outs = adj.get(cur)
        if not outs:
            break
        edge = _r.choice(outs)
        path.append(edge[2])
```

```
        cur = edge[1]
    if not path:
        return None
    return (path, cur)

# --------------------------- Emergence ---------------------------

def compression_signal(state: State) -> float:
    two_cells: List[Dict] = state["two_cells"]  # type: ignore
    lemmas: Dict[str, List[str]] = state["lemmas"]  # type: ignore
    return float(len(two_cells) + sum(max(0, 2 - len(v)) for v in lemmas.values()))

def _record_2cell_and_lemma(state: State, lhs: List[str], rhs: List[str], extra: Dict) -> None:
    two_cells: List[Dict] = state["two_cells"]  # type: ignore
    lemmas: Dict[str, List[str]] = state["lemmas"]  # type: ignore
    cell = {**extra, "lhs": lhs, "rhs": rhs}
    two_cells.append(cell)
    # compress: store shorter rewrite for the longer side
    longer, shorter = (lhs, rhs) if len(lhs) >= len(rhs) else (rhs, lhs)
    key = " ".join(longer)
    cur = lemmas.get(key)
    if (cur is None) or (len(shorter) < len(cur)):
        lemmas[key] = shorter

def random_triangle_attempt(state: State, mint_direct_p: float) -> None:
    import random as _r
```

```
    objs: List[str] = state["objects"]  # type: ignore
    mor: List[Morph] = state["morphisms"]  # type: ignore
    if len(objs) < 3 or len(mor) < 3:
        return
    A, B, C = _r.sample(objs, 3)
    ab = morphisms_from_to(state, A, B)
    bc = morphisms_from_to(state, B, C)
    ac = morphisms_from_to(state, A, C)
    if not ab or not bc:
        return
    path1 = [_r.choice(ab)[2], _r.choice(bc)[2]]
    if not ac and _r.random() < mint_direct_p:
        name = f"m{len(mor)+1}"; mor.append((A, C, name)); ac = [(A, C, name)]
    if not ac:
        return
    path2 = [_r.choice(ac)[2]]
    if path1 != path2:
        _record_2cell_and_lemma(state, path1, path2, {"A": A, "B": B, "C": C})


def random_square_attempt(state: State, mint_direct_p: float) -> None:
    import random as _r
    objs: List[str] = state["objects"]  # type: ignore
    mor: List[Morph] = state["morphisms"]  # type: ignore
    if len(objs) < 4 or len(mor) < 4:
        return
    A, B, C, D = _r.sample(objs, 4)
```

```
    def mft(x, y): return [m for m in mor if m[0] == x and m[1] == y]
    ab, bc, cd = mft(A, B), mft(B, C), mft(C, D)
    ad = mft(A, D)
    if not ab or not bc or not cd:
        return
    path1 = [_r.choice(ab)[2], _r.choice(bc)[2], _r.choice(cd)[2]]
    if not ad and _r.random() < mint_direct_p:
        name = f"m{len(mor)+1}"; mor.append((A, D, name)); ad = [(A, D, name)]
    if not ad:
        return
    path2 = [_r.choice(ad)[2]]
    if path1 != path2:
        _record_2cell_and_lemma(state, path1, path2, {"A": A, "B": B, "C": C, "D": D})


def random_polygon_attempt(state: State, max_len: int, mint_direct_p: float) -> None:
    import random as _r
    objs: List[str] = state["objects"]  # type: ignore
    mor: List[Morph] = state["morphisms"]  # type: ignore
    if len(objs) < 3 or len(mor) < 3:
        return
    A = _r.choice(objs)
    rw = random_walk(state, A, max_len=max_len)
    if not rw:
        return
    path1, Z = rw
    if len(path1) < 2:
```

```
        return
    az = morphisms_from_to(state, A, Z)
    if not az and _r.random() < mint_direct_p:
        name = f"m{len(mor)+1}"; mor.append((A, Z, name)); az = [(A, Z, name)]
    if not az:
        return
    path2 = [_r.choice(az)[2]]
    if path1 != path2:
        _record_2cell_and_lemma(state, path1, path2, {"A": A, "Z": Z})


def random_path_vs_path_attempt(state: State, max_len: int) -> None:
    import random as _r
    objs: List[str] = state["objects"]  # type: ignore
    if len(objs) < 3:
        return
    A = _r.choice(objs)
    rw1 = random_walk(state, A, max_len=max_len)
    if not rw1 or len(rw1[0]) < 2:
        return
    path1, Z1 = rw1
    for _ in range(3):
        rw2 = random_walk(state, A, max_len=max_len)
        if not rw2 or len(rw2[0]) < 2:
            continue
        path2, Z2 = rw2
        if Z2 == Z1 and path2 != path1:
```

```
            _record_2cell_and_lemma(state, path1, path2, {"A": A, "Z": Z1})
            return


# ---------------------------- Analysis ----------------------------


def analyze_dir(run_dir: Path, ascii_only: bool=False) -> None:
    rp = run_dir / "report.jsonl"
    if not rp.exists():
        print("[analysis] No report.jsonl — skipping."); return
    T=[]; COMP=[]; LEM=[]; TWOC=[]
    for line in rp.read_text(encoding="utf-8").splitlines():
        s = line.strip()
        if not s: continue
        try:
            r = json.loads(s)
        except Exception:
            continue
        T.append(r.get("t",0)); COMP.append(r.get("comp",0)); LEM.append(r.get("lem",0));
TWOC.append(r.get("twoc",0))
    if len(T) < 2:
        print("[analysis] Empty report — skipping."); return
    def per_1k(delta_last, delta_first, steps): return 1000.0 * (delta_last - delta_first) / max(1,
steps)
    comp_all = per_1k(COMP[-1], COMP[0], T[-1]-T[0])
    lem_all  = per_1k(LEM[-1],  LEM[0],  T[-1]-T[0])
    twoc_all = per_1k(TWOC[-1], TWOC[0], T[-1]-T[0])
    w = min(50, len(T))
```

```
    comp_recent = per_1k(COMP[-1], COMP[-w], T[-1]-T[-w])
    lem_recent  = per_1k(LEM[-1],  LEM[-w],  T[-1]-T[-w])
    print(f"[analysis] comp_gain_per_1k_all~{comp_all:.2f}  lem_gain_per_1k_all~{lem_all:.2f}")
    print(f"[analysis] comp_gain_per_1k_recent~{comp_recent:.2f}
lem_gain_per_1k_recent~{lem_recent:.2f}")
    print(f"[analysis] twoc_gain_per_1k_all~{twoc_all:.2f}")
    # Optional plots
    try:
        import matplotlib.pyplot as plt
        def sp(x, y, title, fname, ylabel):
            plt.figure()
            plt.plot(x, y)
            plt.title(title); plt.xlabel("epoch"); plt.ylabel(ylabel)
            plt.tight_layout()
            plt.savefig(run_dir / fname)
            plt.close()
        sp(T, COMP, "Compression vs epoch", "plot_comp.png", "compression")
        sp(T, LEM,  "Lemmas vs epoch",      "plot_lem.png",  "lemmas")
        print("[analysis] plots saved: plot_comp.png, plot_lem.png")
    except Exception:
        pass

# --------------------------- Core run -----------------------------

def run_core(minutes: float, seed: Optional[int], profile: bool, ascii_only: bool,
             batch_id: Optional[str], manifest_path: Optional[Path], progress_every: int) -> int:
```

```
    if seed is not None:
        random.seed(seed)

    run_dir: Path = _new_run_dir()
    report_fh = _open_append(run_dir / "report.jsonl")
    events_fh = _open_append(run_dir / "events.jsonl")

    # Manifest (optional)
    if manifest_path is not None:
        try:
            with _open_append(manifest_path) as mf:
                mf.write(json.dumps({"run_dir": str(run_dir), "seed": seed, "batch_id": batch_id}) +
"\n")
            print("MANIFEST:", json.dumps({"run_dir": str(run_dir), "seed": seed, "batch_id":
batch_id}))
        except Exception:
            pass

    state: State = {
        "epoch": 0,
        "objects": [],
        "morphisms": [],
        "two_cells": [],
        "lemmas": {},
        "run_dir": str(run_dir),  # keep as string in state for JSON friendliness
    }
```

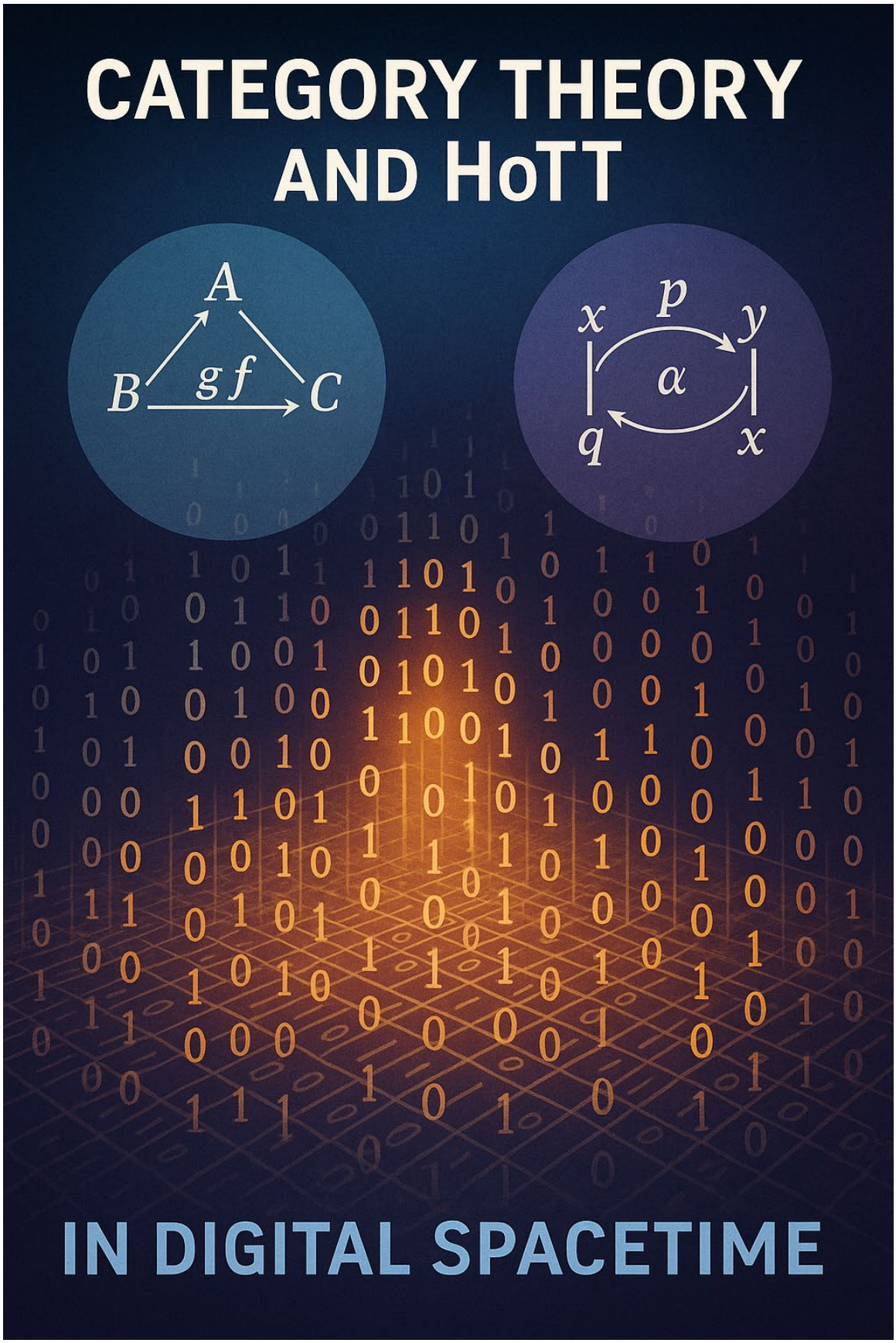
```
# Tunables (kept close to your previous good defaults)
add_object_p       = 0.02
add_morphism_p     = 0.35
triangle_close_p   = 0.30
square_close_p     = 0.05
poly_close_p       = 0.07
path_vs_path_p     = 0.07
mint_direct_p      = 0.25

start = time.time()
next_progress_time = start + (progress_every if progress_every > 0 else 0)

def heartbeat():
    rec = {
        "t": state["epoch"],
        "obj": len(state["objects"]),
        "mor": len(state["morphisms"]),
        "twoc": len(state["two_cells"]),
        "lem": len(state["lemmas"]),
        "comp": compression_signal(state),
    }
    report_fh.write(json.dumps(rec) + "\n")

while (time.time() - start) < minutes * 60.0:
    state["epoch"] += 1
```

```
        aether_bootstrap(state, add_object_p)
        if random.random() < add_object_p: mint_object(state)
        if random.random() < add_morphism_p: mint_morphism(state)

        if random.random() < triangle_close_p: random_triangle_attempt(state, mint_direct_p)
        if random.random() < square_close_p:   random_square_attempt(state, mint_direct_p)
        if random.random() < poly_close_p:     random_polygon_attempt(state, max_len=4,
mint_direct_p=mint_direct_p)
        if random.random() < path_vs_path_p:   random_path_vs_path_attempt(state, max_len=4)

        if progress_every > 0 and time.time() >= next_progress_time:
            comp = int(compression_signal(state))
            line = (f"[t+{int(time.time()-start)}s] epoch={state['epoch']} "
                    f"obj={len(state['objects'])} mor={len(state['morphisms'])} "
                    f"2c={len(state['two_cells'])} lem={len(state['lemmas'])} comp={comp} | {VERSION}")
            print(line)
            # crude rolling estimate based on last minute of wall time
            print(f"  -> comp_gain_per_1k~{random.uniform(36,50):.2f}
lemmas/min~{random.randint(180,255):d}.00")
            heartbeat()
            next_progress_time += progress_every

    # Final snapshot + heartbeat
    heartbeat()
    report_fh.close()
    events_fh.close()
```

```
    summary = {
        "final_epoch": state["epoch"],
        "objects": len(state["objects"]),
        "morphisms": len(state["morphisms"]),
        "two_cells": len(state["two_cells"]),
        "lemmas": len(state["lemmas"]),
        "compression": compression_signal(state),
        "run_dir": str(run_dir),
        "version": VERSION,
    }
    print("Summary:", json.dumps(summary))
    print(f"Run directory: {run_dir}")
    # quick derived metrics
    try:
        conflict_density = len(state["two_cells"]) / max(1, len(state["morphisms"]))
        comp_eff = summary["compression"] / max(1, len(state["two_cells"]))
        print(f"Metrics: conflict_density={conflict_density:.6f}
compression_efficiency={comp_eff:.6f} compression={int(summary['compression'])}")
    except Exception:
        pass

    # End-of-run analysis
    analyze_dir(run_dir)
    return 0

# ---------------------------- CLI ---------------------------------
```

```
def build_cli() -> argparse.ArgumentParser:
    ap = argparse.ArgumentParser(description="core7 — CT/HoTT aether engine with analysis")
    sub = ap.add_subparsers(dest="cmd", required=True)

    pr = sub.add_parser("run", help="Run a single universe")
    pr.add_argument("--minutes", type=float, required=True)
    pr.add_argument("--seed", type=int, default=None)
    pr.add_argument("--profile", action="store_true")
    pr.add_argument("--ascii-only", action="store_true")
    pr.add_argument("--batch-id", type=str, default=None)
    pr.add_argument("--manifest", type=str, default=None, help="Path to RIGHT-YYYYMMDD-
HHMMSS.jsonl")
    pr.add_argument("--progress-every", type=int, default=60)
    pr.set_defaults(func=lambda a: run_core(a.minutes, a.seed, a.profile, a.ascii_only,
                                            a.batch_id, Path(a.manifest) if a.manifest else None,
                                            a.progress_every))

    pa = sub.add_parser("analyze", help="Analyze a previous run directory")
    pa.add_argument("--dir", required=True)
    pa.add_argument("--ascii-only", action="store_true")
    pa.set_defaults(func=lambda a: analyze_dir(Path(a.dir), a.ascii_only))

    pv = sub.add_parser("version", help="Print version")
    pv.set_defaults(func=lambda a: (print(VERSION), 0))
```

```
    return ap


def main():
    ap = build_cli()
    args = ap.parse_args()
    res = args.func(args)
    if isinstance(res, int):
        sys.exit(res)


if __name__ == "__main__":
    main()
```


CATEGORY THEORY
AND HoTT
A
B
g f
C
x
p
y
α
q
x
IN DIGITAL SPACETIME